ECE 492

Spring 2022

Final Project

Final Project Report

Qian, Liyang; Jin, Yihong

liyangq2, yihongj3

Instructor: Raghavendra Kanakagiri

1 Abstract

This report optimizes and analyzes the performance of a parallel algorithm using MPI and OpenMP for TSP problem. The parallel algorithm is developed based on the Simulated Annealing algorithm to find the shortest path that visits n specified locations, starting and ending at the same place and visiting the other n-1 destinations exactly once. This report starts from a sequential Simulated Annealing algorithm and parallelizes it with MPI. Then, we try MPI+OpenMP strategy but do not get an improvement due to the limitation of experiment environment. The performances of different version of codes are compared with each other. Results show that our implementation with MPI provides a dramatic improvement.

2 Idea and Overview

2.1 Introduction

In this project, we decide to use MPI to make the parallelization of TSP . The sequential algorithm of TSP is implemented with the Simulated Annealing algorithm, SA in short, which is used to find the global minimum value of the object function with randomly choosing the solution instead of enumerating solutions and comparing their corresponding object function value. In this problem, we set the ordered array of city positions as the solution to TSP, and the path length accumulated from the distance between two adjacent elements in the ordered array as the object function for SA.

To be specific, firstly, starting temperature, T_0, and end temperature,T_{end}, are set as the threshold to finish the SA algorithm. The system's current temperature, T_{cur}, changes from T_0 to T_{end}, updated by $T_{cur} = T_{cur} * q$. In each value of T_{cur}, there are L iterations. In

each iteration, two random cities in node array will be swapped to generate a new solution of TSP, with corresponding total path length Dis_{new}. Then if the new value is smaller than the current total path length Dis_{cur} of the temperature result, the new solution, array of cities, is accepted as the temperature result,S_{temp}, of TSP. If $Dis_{new} > Dis_{cur}$, the new solution is accepted as the S_{temp} with probability $P(Accept) = e^{df/T_{cur}}$. When $T_{cur} < T_{end}$, algorithm ends, and the S_{temp} is the solution of TSP.

2.2 MPI idea

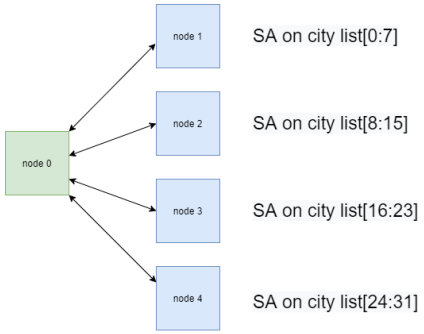
The idea of MPI for this problem is that the whole graph is partitioned into exclusively parts, and each process except the process 0, the master process, will have one part of the graph. Then we do TS algorithm on these parts and send the part solution to process 0, the process returning the final solution to the TSP. After receiving the partial solutions from different processes, the master process gets the final solution by trying different permutation of the partial solutions and then choosing the solution that has the least path length as the final solution.

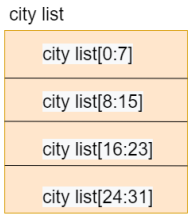
2.3 openMP idea

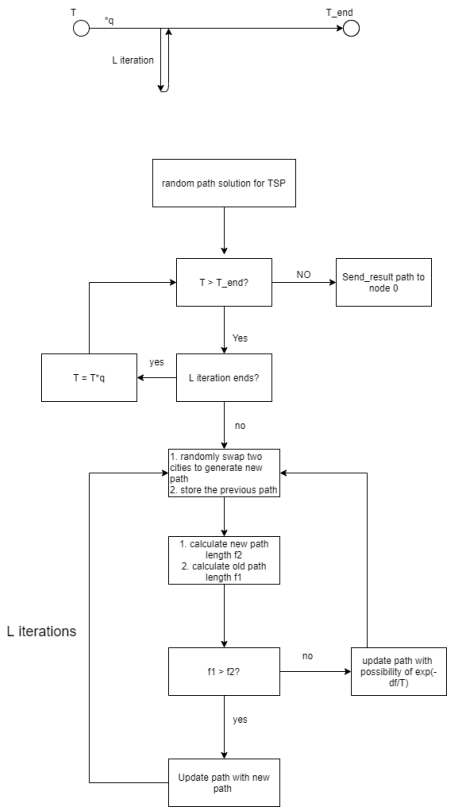
The idea of openMP used in our project is to parallelize the calculation of path length for the TSP solution. In the pre-optimization code, the path length is computed by sequentially summing up the distance in the city list. Therefore, we parallize the code by *parallel for*, which can also be replaced by *openMP reduction*.

2.4 Program graph

Following is the graph for what the whole program looks like:







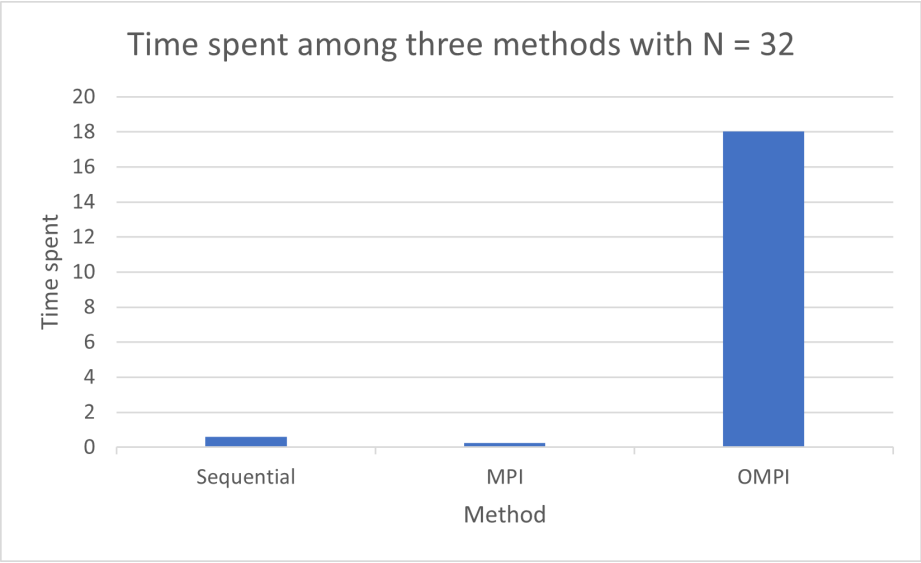
3 Performance graph and Data Analysis

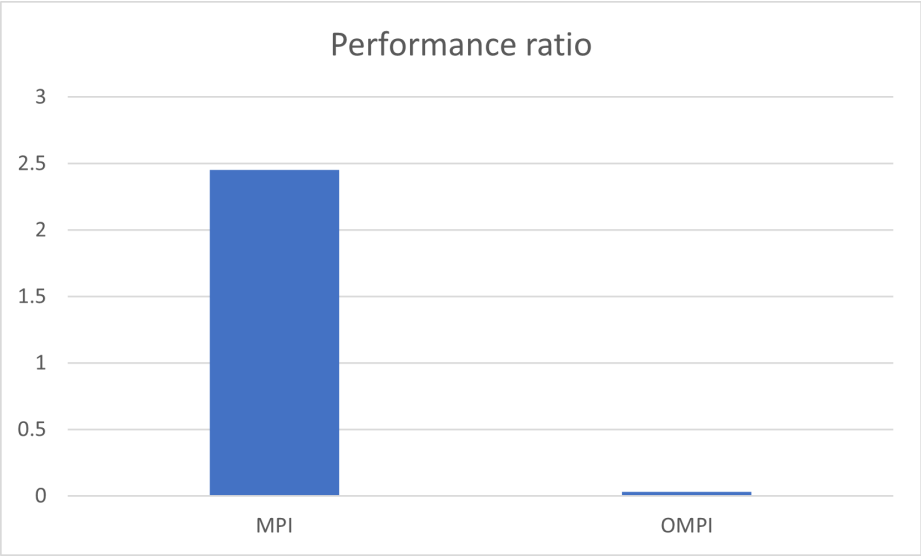
To see the performance of our code, we divide the code test into two parts. In the first part, we compare the performance among **sequential version, MPI version and MPI+openMP version(OMPI)** with the number of cities, N, is 32. In the second part, we compare the performance with N in 32, 64, 128 and 256. The result is below.

The number of processes for MPI is 4, and the threads allocated for openMP is 2.

About the **Test environment**: the test environment machine is the Virtual machine for CS420, which has 6 physical cores and each of the cores contains 1 siblings. Thus, total logical core for the machine is 6, which means each process has only one thread.

3.1 Spent time for each method on Data with size 32

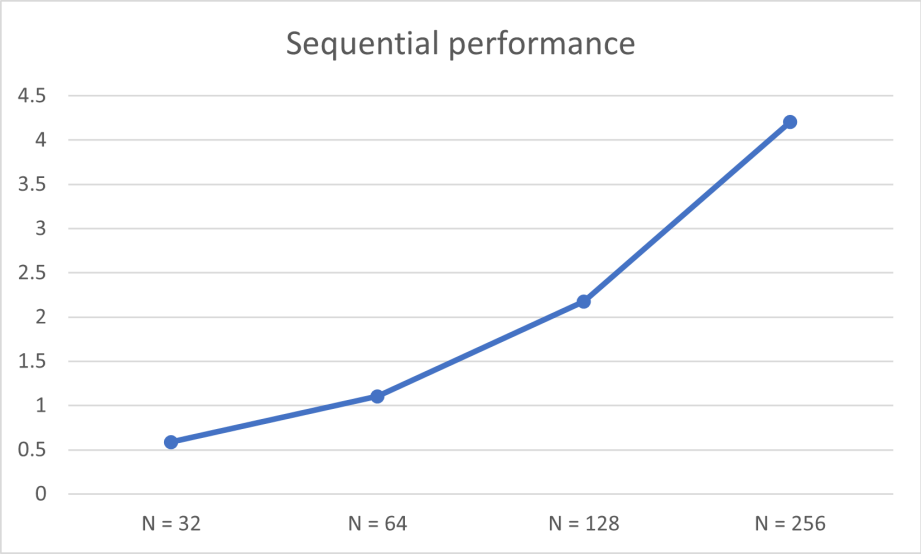




The MPI implementation achieves nearly 2.5 speed up ratio compared to the sequential version. The reason why the ration is not 4 is that employing MPI introduces extra overhead while the data set is small. In the following bench marking, we will show that the speed up ratio will approach 4 if the size of data set is large enough, which means that the extra overhead introduced by employing MPI can be ignored compared to the performance improvement introduced by parallelism.

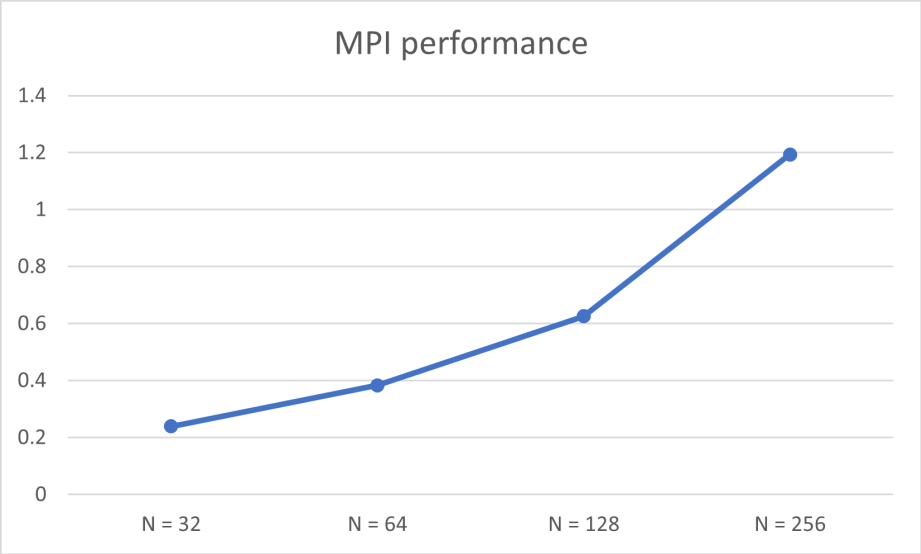
However, The MPI+openMP implementation is even worse than sequential implementation. The reason why it is slow is that in our test environment, each processor has only one core which means that local multi-threading of openMP will be meaningless while it introduces more overhead.

3.2 Graph of spent time with N increasing in Sequential



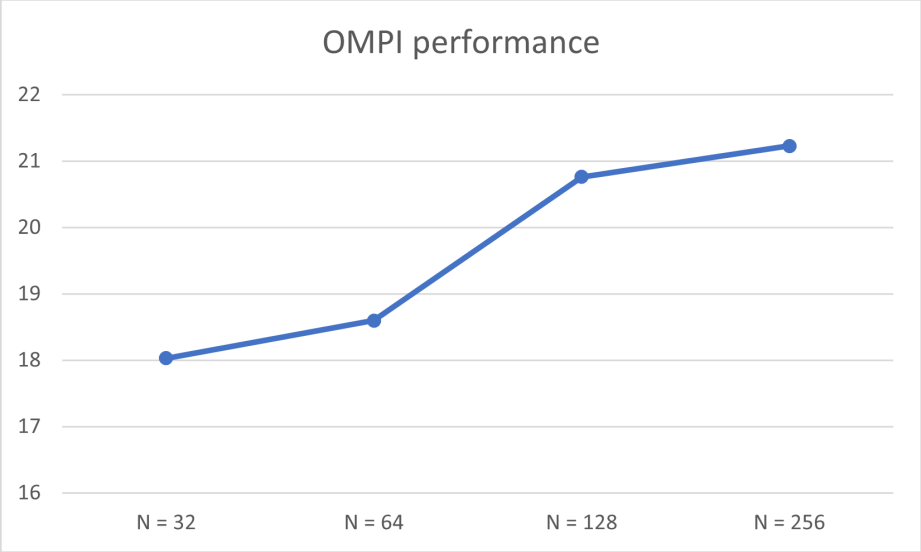
The time required for sequential version execution is linearly related to size of the data set.

3.3 Graph of spent time with N increasing in MPI



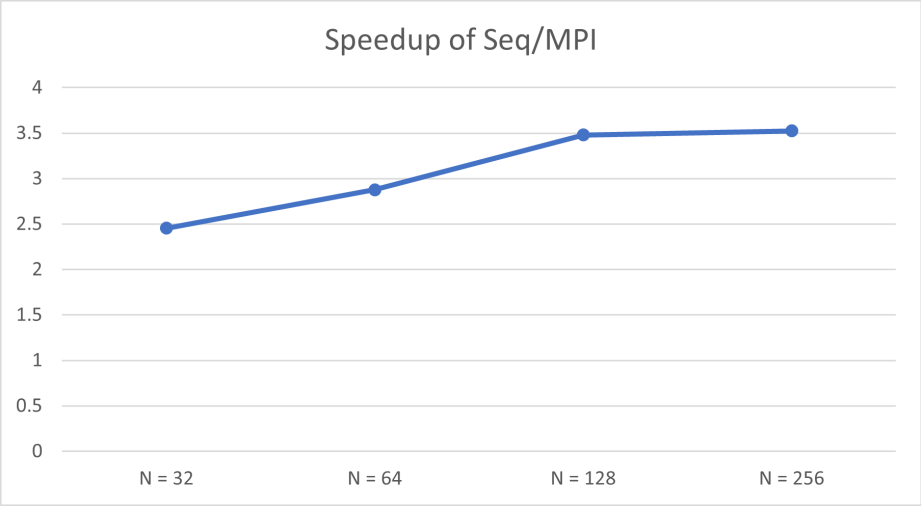
The time required for MPI version execution increases while size of the data set increases. However the relation between them is not linear. The increment of MPI version execution time is slower than the increment of size of the data set.

3.4 Graph of spent time with N increasing in OMPI



According to the graph above, the time spent by OMPI method continuous increases with N increasing. There is a big time increment when the data size varies from 64 to 128.

3.5 Graph of Speed up with N increasing between sequential and MPI



According to the graph above, the speed up for MPI method to sequential method continuously increases with N increasing, and the speedup value converges to 4, which means the compute task takes more part in the program executing time.

4 Running instruction

In the folder of the project, we provide three shell files to compile and run our code.

The *sequential-run.sh* is the file that sequential code.

The *mpi-run.sh* is the MPI optimization of the code.

The *ompi-run.sh* is the MPI+openMP(OMPI) optimization.